The Advent of Recursion in Programming, 1950s-1960s

Edgar G. Daylight**

Institute of Logic, Language, and Computation,
University of Amsterdam, The Netherlands
egdaylight@yahoo.com

Abstract. The term 'recursive' has had different meanings during the past two centuries among various communities of scholars. Its historical epistemology has already been described by Soare (1996) with respect to the mathematicians, logicians, and recursive-function theorists. The computer practitioners, on the other hand, are discussed in this paper by focusing on the definition and implementation of the ALGOL60 programming language. Recursion entered ALGOL60 in two novel ways: (i) syntactically with what we now call BNF notation, and (ii) dynamically by means of the recursive procedure. As is shown, both (i) and (ii) were introduced by linguistically-inclined programmers who were not versed in logic and who, rather unconventionally, abstracted away from the down-to-earth practicalities of their computing machines. By the end of the 1960s, some computer practitioners had become aware of the theoretical insignificance of the recursive procedure in terms of computability, though without relying on recursive-function theory. The presented results help us to better understand the technological ancestry of modern-day computer science, in the hope that contemporary researchers can more easily build upon its past.

Keywords: recursion, recursive procedure, computability, syntax, BNF, ALGOL, logic, recursive-function theory

1 Introduction

In his paper, *Computability and Recursion* [41], Soare explained the origin of *computability*, the origin of *recursion*, and how both concepts were brought together in the 1930s by Gödel, Church, Kleene, Turing, and others. I summarize part of Soare's paper below, omitting most of his bibliographical references.

The origin of computability can be briefly described by mentioning the Babylonians, Euclid, al-Khwarizmi, Pascal, Leibniz, and Babbage. Leibniz searched for a universal language and a calculus of reasoning. Babbage invented his programmable Analytic Engine and formulated what we now call Babbage's Thesis, which states that "the whole of the development and operations of analysis are now capable of being executed by machinery" [13, p.57].

** Also known as Karel Van Oudheusden

The notion of recursion dates back to the late-19th century mathematicians Dedekind and Peano, who used the principle of defining a function by induction. This principle played an important role in the foundations of mathematics (cf. the work of Skolem, Hilbert, Ackermann, Gödel, and Péter) and was only much later called 'primitive recursion' by Péter (1934) and Kleene (1936).

To describe how both the concepts of computability and recursion were brought together, Soare mentioned Hilbert's work, the Entscheidungsproblem, and Gödel's incompleteness theorems. Gödel knew that not all effectively calculable functions were included in his 'recursive functions' (1931), which would therefore later be called 'primitive recursive functions'. Based on a suggestion from Herbrand, Gödel searched for a more general class of functions, resulting in his 'general recursive functions' (1934), later to be called 'recursive functions'.

Only in 1936, in a paper of Church [8], did 'recursively' connote 'effectively' or 'computably' for the first time. Later work of Kleene and Post reinforced this connotation but only inside their newly built community of recursive-function theorists. Neither Gödel nor Turing ever used the adjective 'recursive' to connote 'computable'. And, most people outside the subject, such as the computer practitioners addressed in this paper and mathematicians, have never associated 'recursive' with 'computable' or 'decidable'.

Soare's historical epistemology of the term 'recursive', summarized above, illustrates nicely the different communities of scholars. In this paper, the meaning of the term 'recursive' is studied further but with respect to the computer practitioners of the late 1950s and early 1960s. Starting with an historical context (Section 2), syntactic recursion is described (Section 3), followed by dynamic recursion (Section 4). Both forms of recursion were introduced by computer programmers who, like many of their colleagues, were not acquainted with recursive-function theory, let alone logic in general (Section 5).

2 Historical Context

With the advent of the programmable computing machine, a new community of numerical analysts emerged. Unlike e.g., the logicians and the electrical engineers, the numerical analysts, by their very profession, took programming seriously [20, p.3]. Several of them gradually became more involved in seeking specific techniques to overcome the tediousness in programming their machines. Two such, and important, techniques were what we now call Backus-Naur-Form (`BNF`) notation and the recursive procedure. Both entered the arena of programming languages via the `ALGOL60` language [18, 19].

The `ALGOL60` Language

In October 1955, various German and Swiss mathematicians had come together in Darmstadt, Germany, to attend a meeting, called Elektronische Rechenmaschinen und Informationsverarbeitung. It marked the beginning of an international

effort involved in creating a universal programming language. Several participants of the meeting stressed the need for one *universal* and *machine-independent algorithmic* language [5, p.5]. Programs in that language were meant to allow people to communicate algorithms with each other without having to execute them on a machine [6, p.139]. The adjective *universal* referred to the aspiration that everybody would communicate with each other in the same algorithmic language. The *machine independence* expressed the desire that the language would be designed without having a specific machine in mind [5, p.6]. Of equal importance is the adjective *algorithmic*. It emphasized the fact that numerical computations were intended to be the main (if not the only) application domain of the language [25, p.101].

By 1958, the Swiss and Germans were collaborating with the Americans by holding an ACM[1]-GaMM[2] meeting in Zürich. The chosen name for the universal programming language was initially `IAL` (International Algorithmic Language), later denoted as `ALGOL58`, but would by January 1960 change into `ALGOL` (Algorithmic Language) [5, p.31], and denoted as `ALGOL60` in this text. As languages, `ALGOL58` and `ALGOL60` would be drastically different [5, p.35].

This "`ALGOL` Effort" would quickly become more international: e.g., the Dutch Edsger W. Dijkstra and the Dane Peter Naur joined. The latter became the editor of the `ALGOL60` report [2], a document that became the standard for defining programming languages [5, p.35] for several decades[3]. After the publication of that report, Naur also initiated an `ALGOL` Bulletin, which served the purpose of discussing properties of `ALGOL` and promoting its use as a programming language [35, p.6].

The `FORTRAN` System

Prior to joining the `ALGOL` Effort and contrary to the Europeans, the Americans already had several programming systems. One of them was `FORTRAN` (`FORmula TRANslator`), invented by John Backus and his team. As early as December 1953, Backus had proposed the `FORTRAN` project to his boss at IBM [4]. In contrast to `ALGOL60`, `FORTRAN` became a de facto standard programming language for scientific computing [5, 38, p.11, p.525]. However, while `ALGOL60` was machine independent, `FORTRAN` had six machine dependent language constructs [5, p.15].

Compared to other existing programming systems of the 1950s, `FORTRAN` was, in hindsight, the first high-level system that met two seemingly contrasting requirements. First, a `FORTRAN` program could be translated into machine code at a sufficiently low cost. Second, the obtained machine code was sufficiently economical in comparison to code that was hand written by an expert machine-level programmer. To meet these requirements, Backus and his team focused on the design of the translator and *not* on that of the language [38, 40, p.525, p.233].

[1] Association for Computing Machinery
[2] Gesellschaft für Angewandte Mathematik und Mechanik
[3] Even today, `ALGOL`-like programming languages are used in industry (e.g., C, Java), and studied in certain branches of theoretical computer science (e.g., [26, 27]).

<digit>	:= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<integer>	:= <digit> | <integer><digit>
<realPart>	:= <integer> | <integer>.<integer>
<real>	:= <realPart> | +<realPart> | −<realPart>

Table 1. An example in Backus Naur Form.

After his FORTRAN years, Backus participated in the ALGOL Effort and made a significant contribution to formal syntax, as described next.

3 Syntactic Recursion: BNF Notation

Prior to 1959, computer practitioners described syntax informally, e.g., in verbose English. As an example, consider the definition: A real number is

> any sequence of decimal digits with a decimal point preceding or intervening between any 2 digits or following a sequence of digits, all of this optionally preceded by a plus or minus sign.

The previous passage is similar to how a real number in IAL was defined syntactically[4]. By including machine-dependent constants, which express the finite-storage limitations of the machine, the previous definition can be extended to:

> The number must be less than 10^{38} in absolute value and greater than 10^{-38} in absolute value.

The previous two passages, together, constitute the original definition of a real number in FORTRAN [16].

In short, both FORTRAN and IAL were defined informally. FORTRAN was defined with and IAL was defined without finite-storage limitations in mind. The informal definitions were ambiguous, incomplete, and often lengthy: FORTRAN's and IAL's syntax were very cumbersome to use in practice [5, p.26,27].

In 1959 at the IFIP congress in Zürich, Backus came to the rescue. Inspired by logic but not understanding much of it himself[5], Backus introduced a formal notation not unlike Post's production rules [18]. However, Backus's paper [1] almost went unnoticed; it was Naur who grasped its potential and, who, after making some small but important modifications, used it to define ALGOL60's syntax. The notation is therefore called Backus Naur Form [3, 18].

Continuing with our running example of the real numbers, the BNF equivalent of the first passage, presented above, is depicted in Table 1. With | denoting logical *or*, Line 1 expresses that a digit is either 0 or 1 or 2 or ... or 9. Line 2, in turn, *recursively* defines an integer to be either a digit or an integer concatenated

[4] For the actual definition in IAL, see [30, p.9].
[5] As explained by Backus himself in [39, p.17].

with a digit. Line 3 defines the real part of a real number to either be an integer or an integer followed by a decimal point and another integer. Finally, Line 4 defines a real number to have no sign, a plus sign, or a minus sign.

Note that Table 1 is only the BNF equivalent of the first passage, presented above. The finite-storage limitations of the machine (cf. the second passage) can not be expressed concisely in BNF notation. Indeed, the syntactic recursion, exemplified in Line 2 in Table 1, is what made BNF notation so concise: Line 2 allows an arbitrarily large but finite integer to be written down in ALGOL60 and, hence, also integers that simply could not fit in every computer's memory!

Backus's conceptual leap of abstracting away the computing machine's finite limitations cannot be stressed enough. Unlike his work during the FORTRAN years, where he focused on the design of the translator to obtain efficient machine code, Backus's abstraction allowed him to *solely* focus on the language. On the one hand, Backus was of course aided by IAL's abstraction of finite storage. On the other hand, as we shall see, many computer practitioners did not let go of machine-specific features while designing a programming language.

4 Dynamic Recursion: the Recursive Procedure

During the early 1960s, there were research groups (of computer practitioners) that were mainly led by linguistic ideals and there were groups that were primarily led by machine-related features. To illustrate this, emphasis will be laid below on Dijkstra's linguistic views[6] towards programming-language design, which he described in his 1963 paper [10], i.e., three years after ALGOL60 had already been defined in [2].

Dijkstra's Linguistic Ideology

In order to discuss programming-language design, Dijkstra first suggested considering any English text that respects five restrictions:

1. Words of more than 15 letters are forbidden.
2. The total number of letters of three consecutive words may not be greater than 40.
3. Sentences of more than 60 words are not allowed.
4. In one and the same sentence, the same word may not be used twice as a subject.
5. A list of 2000 words is given and each word in that list may not be used.

Dijkstra remarked that (i) the readability of any text respecting 1-5 is not necessarily hindered and (ii) one can read such a text while being completely ignorant of the existence of restrictions 1-5. However, constructing a *correct* English text can become very problematic if more restrictions are added to the above list or especially if they impose highly implicit conditions [10, p.31].

[6] I have no evidence to suggest that Dijkstra was influenced by Chomsky.

Dijkstra then distinguished between a natural language and a programming language, by considering two scenarios. In Scenario 1, a speaker communicates with a listener by talking in English. In Scenario 2, a programmer communicates with a computer by programming. Dijkstra explained the difference between both scenarios: a listener is rather unpredictable in his reactions, while a computer can, essentially, be completely understood and, hence, be predictable. To exploit this advantage that a computer can have over a human, Dijkstra stressed the importance of avoiding an unnecessarily complicated language, and expressed his disappointment with `ALGOL60` in this respect [10, p.33,34].

Dijkstra subsequently applied his ideology to `ALGOL60`. Just as in his English-language example, Dijkstra wanted to reduce the number of 'unnecessary' restrictions in `ALGOL60`. To do so, he presented examples in which he advocated for *dynamic* instead of static constructions since they make the language more systematic and powerful. For example, concerning arrays, Dijkstra suggested removing the explicit specification of an array's subscript bounds, since they become determined at run time any way. Omitting the array bounds, in turn, resulted in more freedom, for, now there was no reason to restrain an array to being rectangular, it could just as well be triangular, etc. [10, p.36].

Dijkstra's ideology led him to the *extreme* of omitting all type indications and, hence, transferring all the type checking to the run-time system [10, p.36], which, as many observed, would have a negative effect on computation time —an observation that Dijkstra did not contradict, cf. [9, 10, p.312, p.41]. These references also show that, according to Dijkstra, generalization of a programming language allowed for simplification in compiler building and this would in the long term prevail over the short-term efficiency problems that concerned many computer practitioners.

Dijkstra's Ideology Led to Recursive Procedures

In his pursuit for a general language, Dijkstra advocated for run-time constructions (cf. above). In terms of procedure calls, he had been no different in 1959–60, as described below.

During the `ALGOL` Effort, the Germans Samelson and Bauer were strong proponents for what we today call *static* memory management. Their `ALCOR` compiler of the `ALGOL60` language statically allocated all procedures (i.e., prior to execution). This implementation choice, which they took in the interest of efficiency, forced them to restrict the way `ALGOL60` was used by disallowing procedures to call other procedures and, hence, recursive procedure calls in particular. As the English-language example has illustrated, it is no surprise that Dijkstra opposed Samelson and Bauer's language restriction: Dijkstra wanted any procedure to be able to call any other procedure in an `ALGOL60` program [9, 36].

Dijkstra's quest to generalize led him to use the well-known concept of a stack (i.e., a continuous portion of computer memory) as a *run-time* object, rather than a mere compile-time object as was the case in Samelson and Bauer's `ALCOR` compiler. In current-day terminology: Dijkstra proposed to use *dynamic* memory management to implement his general procedure calls [9, 36]. By doing

so, he in 1960, together with his colleague Zonneveld, became internationally the first to build an ALGOL60 compiler that could handle recursive procedures [19].

During the 1960s, recursive procedures remained controversial, however, and Dijkstra was one of its few strong advocates. His linguistic views were in sharp contrast to those led by specific machine features, as Strachey's words from 1962 illustrate:

> I think the question of simplifying or reducing a language in order to make the object program more efficient is extremely important. I disagree fundamentally with Dijkstra, about the necessity of having everything as general as possible in all possible occasions as I think that this is a purely theoretical approach [...] [29, p.368]

To summarize, many computer practitioners indirectly took specific machine features into account while discussing a machine-independent language, such as ALGOL60. Since they believed that recursive procedures were inefficient to execute, they wanted to restrict the way in which ALGOL60 was used by eschewing recursive procedures [29]. This bottom-up approach (i.e., from machine to language) is in sharp contrast to Dijkstra's top-down approach in which language was studied before any machine-related features were addressed [10].

5 Theoretical Insignificance of the Recursive Procedure

Backus, Naur, and Dijkstra had an aptitude for linguistics and were not versed in logic and certainly not in recursive-function theory (cf. [39, 12, 14, p.17, p.346, p.298]). In fact, the great majority of computer practitioners of the 1960s were not aware of the theoretical insignificance of the recursive procedure in terms of computability, as is illustrated by Sammet's 1969 book:

> Recursive procedures were introduced by ALGOL. They certainly should be considered a significant contribution to the technology, but it is not clear how great a one. The advocates of this facility claim that many important problems cannot be solved without it; on the other hand, people continue to solve numerous important problems without it and even in a few cases manage to handle (sometimes in an awkward way) some of the problems which the recursion proponents claim cannot be done. [37, p.193].

Given that ALGOL60's definition allows one to express potentially unbounded while loops [2], it immediately follows from Kleene's normal form theorem[7] that recursive procedures are *not* needed. That is, the expressive power of ALGOL60 is not reduced by discarding recursive procedures.

[7] Presented in Minsky's popular 1967 book [23, p.184], but already published by Kleene in 1936 in [17]. See also Harel's [15] in which he explains the relationship between Kleene's normal form theorem and the *while* construct.

Dijkstra and some others, however, did realize by the late 1960s that the recursive procedure was indeed superfluous in terms of computability, though they based themselves on the technically sophisticated work of Böhm and Jacopini [7] which did not depend on Kleene's normal form theorem[8], as explained in [15].

In general, it is safe to say that recursive-function theory did not influence the computer practitioners during the 1960s. Or did it? In a two-page 1960 letter [33] the recursive-function theorist Rice explained exactly the theoretical significance of "general recursive functions" and its practical implications in programming. Nevertheless, I speculate that his message went unnoticed to many. In fact, he tried to convey the very same message five years later [34].

6 Related Work

Some additional observations are mentioned here. First, the practical usefulness of the recursive procedure was not understood by most computer practitioners during the `ALGOL` Effort [25, 31, p.160, p.86]. Second, it was John McCarthy who urged to add the recursive procedure to the `ALGOL60` language. This was after he had already introduced recursion in his `LISP` language [39, p.27]. Although inspired by Church's lambda calculus, he was not aware of the relevance of `LISP`'s recursion in terms of computability [22, p.176,190]. Finally, it should also be noted that Dijkstra was not the first inventor of the run-time stack principle, i.e., of a technique to implement recursive procedure calls (cf. [28, p.37,38]).

In his Ph.D. thesis, Priestley has examined the influence of logic on the evolution of notations for expressing computer programs. He argues that logic was deliberately employed by designers to obtain a model for theoretically understanding programming languages [32, p.3]. This paper, however, along with my thesis [28], suggests otherwise. Priestley's account is nevertheless impressive: his Chapter 6, for instance, describes how McCarthy, during the 1960s, led a community of computer practitioners from compiler-based semantics to denotational semantics. His observations show that it took several years for the community as a whole to switch from bottom-up to top-down reasoning.

The historian Mahoney has stressed the diversity among the creators and practitioners of what we now call computer science. Different groups of people saw different possibilities in computing [21]. In this paper, however, I have solely focused on some of those involved in the `ALGOL` Effort. Other views towards computing are for instance Lehmer's and Von Neumann's, as explained in [24]. Finally, the reader may wish to study De Beer's thesis [5] as a secondary source: it covers `ALGOL60` in greater generality, compared to this paper.

7 Conclusions

The epistemology of the term 'recursive' was very different for the linguistically-inspired computer practitioners such as Backus, Naur, and Dijkstra, compared to

[8] In the case of Dijkstra, see his 1968 paper [11] in which he describes the work of Böhm and Jacopini [7].

the logicians (e.g., Gödel), the recursive-function theorists (e.g., Kleene), and the many computer practitioners who primarily reasoned bottom-up (e.g., Samelson, Bauer, Strachey) during the early 1960s. Backus's and Naur's syntactic recursion required the conceptual leap of abstracting away the finite limitations of practical computing machines. For Dijkstra, inclusion of the recursive procedure in the ALGOL60 language led to a simplification in terms of language design and subsequent compiler building; he did not consider it to be a hindrance in terms of computation time as many others did. By making the conceptual leap from compile-time to run-time usage of the stack, Dijkstra was able to implement recursive-procedure calls in a relatively simple manner, although he was not the first to do so. Dijkstra and many others did not, however, see any immediate practical value in using the recursive procedure during programming, nor were they aware of the theoretical insignificance of the recursive procedure in terms of computability when ALGOL60 was designed and implemented.

References

1. J.W. Backus, 'The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM Conference', IFIP, pp.125-132, 1959.
2. J.W. Backus et al, Edited by P. Naur, 'Report on the algorithmic language ALGOL60', CACM 3:5, pp.299-314, 1960.
3. J.W. Backus, 'Programming in America in the 1950s – Some Personal Impressions', in Metropolis, N., Howlett, J. and Rota, Gian-Carlo, editors, *A History of Computing in the twentieth century*, pp.125-135, Academic Press, 1980.
4. J.W. Backus, 'The History of FORTRAN I, II, and III' and the corresponding transcripts of: presentation, discussant's remarks, question and answer session, in History of Programming Languages, R.L. Wexelblat, (Ed.), Academic Press, 25-70, 1981.
5. H.T. de Beer, *The History of the ALGOL Effort*, Masters Thesis, Tech. Univ. Eindhoven, Department of Mathematics and Computer Science, August 2006. *http://heerdebeer.org/ALGOL*
6. A. v.d. Bogaard, 'Stijlen van programmeren 1952-1972', Studium 2, 128-144, 2008.
7. C. Böhm, G. Jacopini, 'Flow diagrams, Turing machines, and languages with only two formation rules', C.ACM 9, 5 pp.366-371, May 1966.
8. A. Church, 'An Unsolvable Problem of Elementary Number Theory', The American Journal of Mathematics, Vol.58, pp.345-363, 1936.
9. E.W. Dijkstra, 'Recursive Programming', Num. Mathematik 2, pp.312-318, 1960.
10. E.W. Dijkstra, 'On the Design of Machine Independent Programming Languages', in Annual Review in: R. Goodman, ed, Automatic Programming 3, pp.27-42, 1963.
11. E.W. Dijkstra, 'Go To Statement Considered Harmful', Letters to the Editor, CACM, Vol. 11 (2), pp.147-148, 1968.
12. E.W. Dijkstra, 'EWD1308: What led to "Notes on Structured Programming"', 2002.
13. R. Gandy, 'The confluence of ideas in 1936', In: Herken, pp.55-111, 1988.
14. D. Gries, *The Science of Programming*, Springer-Verlag, New York Inc., 1981.
15. D. Harel, 'On Folk Theorems', CACM, Vol.23, Nr.7, pp.379-388, 1980.
16. IBM, Programming Research Group, 'Preliminary Report – Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN', Technical Report, New York: IBM, 1954.

17. S.C. Kleene, 'General recursive functions of natural numbers', Math. Annalen 112, pp.727-742, 1936.
18. D.E. Knuth, 'Backus Normal Form vs. Backus Naur Form', Letters to the Editor of the C.ACM, pp. 735-736, Vol.7, No.12, December, 1964.
19. F.E.J. Kruseman Aretz, *The Dijkstra-Zonneveld* `ALGOL60` *compiler for the Electrologica X1 (historical note SEN, 2)*, SEN-N0301, ISSN 1386-369X, 2003.
20. M.S. Mahoney, 'The Roots of Software Engineering', An expanded version of a lecture presented at CWI on February 1990. CWI Quarterly 3, 4, pp.325-334, 1990.
21. M.S. Mahoney, 'The histories of computing(s)', Interdisciplinary Science Reviews, Vol.30, No.2, pp.119-135, 2005.
22. J. McCarthy, 'History of `LISP`' and the transcripts of: presentation, discussant's remark, question and answer session, in *History of Programming Languages*, R.L. Wexelblat, (Ed.), New York: Academic Press, 173-195, 1981.
23. M. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc. 1967.
24. L. De Mol, 'Doing Mathematics on the ENIAC. Von Neumann's and Lehmer's different visions.', Mathem. Practice and Dev. throughout History. Proc. of the 18th Noverbertagung on the History, Philosophy and Didactics of Mathematics, 2009.
25. P. Naur, 'The European side of the last phase of the development of `ALGOL60`', in *History of Programming Languages*, R.L. Wexelblat, (Ed.), New York: Academic Press, 92-138, 147-170, 1981.
26. P. O'Hearn, R. Tennent, *Algol-like Languages (Progress in Theoretical Computer Science)*, Birkhäuser Boston, December, 1996, ISBN: 0817639373.
27. C.-H.L. Ong, 'Observational Equivalence of 3rd-Order Idealized Algol is Decidable', LICS, 17th Annual IEEE Symposium on Logic in Computer Science, pp.245, 2002.
28. K. Van Oudheusden –alias Edgar G. Daylight, *The Advent of Recursion & Logic in Computer Science*, Masters Thesis in Logic, University of Amsterdam, November 17th, 2009. *http://www.illc.uva.nl/Publications/ResearchReports/MoL-2009-12.text.pdf*
29. Panel Discussion: 'Efficient Processor Construction', p.363-381 in Proc. of the Symposium Symbolic Languages in Data Processing, Rome, March 26-31, 1962.
30. A.J. Perlis, K. Samelson, 'Preliminary Report: International Algebraic Language', CACM 1:12, 1958.
31. A.J. Perlis, 'The American side of the last phase of the development of `ALGOL`', in *History of Programming Languages*, Richard L. Wexelblat, (Ed.), New York: Academic Press, 75-91, 139-147, 1981.
32. P.M. Priestley, *Logic and the Development of Programming Languages, 1930-1975*, University College London, PhD thesis, May, 2008.
33. H.G. Rice, 'Letters to the Editor', CACM, L12-L13, Sep., 1960.
34. H.G. Rice, 'Recursion and Iteration', CACM, Vol. 8, Number 2, Feb., 1965.
35. H. Rutishauser, *Description of* `ALGOL60`, Handbook for Automatic Computation, Vol. 1, part a. Berlin and New York: Springer-Verlag, 1967.
36. K. Samelson, F. Bauer, 'The ALCOR project', in: Gordon and Breach, editors, Symbolic languages in data processing: Proc. of the Symp. organized and edited by the Int. Computation Center, Rome, 26-31, pp.207-218, New York, 1962.
37. J.E. Sammet, *Programming Languages: History and Fundamentals*, 1969.
38. J.E. Sammet, 'History of IBM's Technical Contributions to High Level Programming Languages', IBM J. Res. Develop. Vol. 25, No. 5, Sept., 1981.
39. D. Shasha, C. Lazere, *Out of their minds: The Lives and Discoveries of 15 Great Computer Scientists,* Copernicus, Springer-Verlag, 1995.
40. R. Slater, *Portraits in Silicon*, MIT Press, First edition 1989, third printing, 1992.
41. R.I. Soare, 'Computability and recursion', Bul. of Symb. Log. 2, pp.284-321, 1996.